



Curso de Shell Script

Papo de botequim

Parte final

A conversa está boa, mas uma hora eles tem que sair do bar. Na última parte do nosso papo, falamos sobre pipes e sincronização entre processos.

POR JÚLIO CEZAR NEVES

E aí rapaz, tudo bom?

– Beleza. Você se lembra de que da última vez você me pediu para fazer um programa que imprimisse dinamicamente, no centro da tela, a quantidade de linhas e colunas de um terminal sempre que o tamanho da janela variasse? Pois é, eu até que fiz, mas mesmo depois de quebrar muito a cabeça a aparência não ficou igual.

– Não estou nem aí para a aparência, o que eu queria é que você exercitasse o que aprendemos. Me dá a **listagem 1** pra eu ver o que você fez.

– Perfeito! Que se dane a aparência, depois vou te ensinar uns macetes para melhorá-la. O que vale é que o programa está funcionando e está bem enxuto.

– Pôxa, e eu que perdi o maior tempo tentando descobrir como aumentar a fonte...

– Deixe isso para lá! Hoje vamos ver umas coisas bastante interessantes e úteis.

Listagem 1: tamtela.sh

```
01 #!/bin/bash
02 #
03 # Coloca no centro da tela, em video reverso,
04 # a quantidade de colunas e linhas
05 # quando o tamanho da tela eh alterado.
06 #
07 trap Muda 28 # 28 = sinal gerado pela mudanca no tamanho
08             # da tela e Muda eh a funcao que fara isso.
09
10 Bold=$(tput bold) # Modo de enfase
11 Rev=$(tput rev) # Modo de video reverso
12 Norm=$(tput sgr0) # Restaura a tela ao padrao default
13
14 Muda ()
15 {
16     clear
17     Cols=$(tput cols)
18     Lins=$(tput lines)
19     tput cup $((Lins / 2)) $(((Cols - 7) / 2)) # Centro da tela
20     echo $Bold$Rev$Cols X $Lins$Norm
21 }
22
23 clear
24 read -n1 -p "Mude o tamanho da tela ou tecle algo para terminar "
```

Dando nomes aos canos

Um outro tipo de pipe é o *named pipe*, que também é chamado de FIFO. FIFO é um acrônimo de *First In First Out* que se refere à propriedade em que a ordem dos bytes entrando no pipe é a mesma que a da saída. O *name* em named pipe é, na verdade, o nome de um arquivo. Os arquivos tipo named pipes são exibidos pelo comando `ls` como qualquer outro, com poucas diferenças, veja:

```
$ ls -l pipe1
prw-r-r-- 1 julio dipao 0 Jan 22 23:11 pipe1
```

o `p` na coluna mais à esquerda indica que `fifo1` é um named pipe. O resto dos bits de controle de permissões, quem pode ler ou gravar o pipe, funcionam como um arquivo normal. Nos sistemas mais modernos uma barra vertical (`|`), ou pipe, no fim do nome do arquivo é outra dica e, nos sistemas LINUX, onde o `ls` pode exibir cores, o nome do arquivo é escrito em vermelho por padrão.

Nos sistemas mais antigos, os named pipes são criados pelo utilitário *mknod*, normalmente situado no diretório */etc*. Nos sistemas mais modernos, a mesma tarefa é feita pelo *mkfifo*, que recebe um ou mais nomes como argumento e cria pipes com esses nomes. Por exemplo, para criar um named pipe com o nome *pipe1*, digite:

```
$ mkfifo pipe1
```

Como sempre, a melhor forma de mostrar como algo funciona é dando exemplos. Suponha que nós tenhamos criado o named pipe mostrado anteriormente. Vamos agora trabalhar com duas sessões ou dois consoles virtuais. Em um deles digite:

```
$ ls -l > pipe1
```

e em outro faça:

```
$ cat < pipe1
```

Voilà! A saída do comando executado no primeiro console foi exibida no segundo. Note que a ordem em que os comandos ocorreram não importa.

Se você prestou atenção, reparou que o primeiro comando executado parecia ter "pendurado". Isto acontece porque a outra ponta do pipe ainda não estava conectada, e então o sistema operacional suspendeu o primeiro processo até que o segundo "abrisse" o pipe. Para que um processo que usa pipe não fique em modo de espera, é necessário que em uma ponta do pipe haja um processo "falante" e na outra um "ouvinte". No exemplo anterior, o *ls* era o "falante" e o *cat* era o "ouvinte".

Um uso muito útil dos named pipes é permitir que programas sem nenhuma relação possam se comunicar entre si. Os named pipes também são usados para sincronizar processos, já que em um determinado ponto você pode colocar um processo para "ouvir" ou para "falar" em um determinado named pipe e ele daí só sairá se outro processo "falar" ou "ouvir" aquele pipe.

Você já deve ter notado que essa ferramenta é ótima para sincronizar processos e fazer bloqueio em arquivos de forma a evitar perda/corrupção de dados devido a atualizações simultâneas (a famosa *concorrência*). Vamos ver alguns exemplos para ilustrar estes casos.

Sincronização de processos

Suponha que você dispare paralelamente dois programas (processos), chamados *programa1* e *programa2*, cujos diagramas de blocos de suas rotinas são como mostrado na **tabela 1**. Os dois processos são disparados em paralelo e, no bloco 1 do programa1, as três classificações são disparadas da seguinte maneira:

```
for Arq in BigFile1 BigFile2 BigFile3
do
  if sort $Arq
  then
    Manda=va
  else
    Manda=pere
  break
fi
done
echo $Manda > pipe1
[ $Manda = pere ] &&
{
  echo Erro durante a classificação dos arquivos
  exit 1
}
...
```

Assim sendo, o comando *if* testa cada classificação que está sendo efetuada. Caso ocorra qualquer problema, as classificações seguintes serão abortadas, uma mensagem contendo a string *pere* é enviada pelo pipe1 e programa1 é descontinuado com código de saída sinalizando um encerramento anormal.

Enquanto o programa1 executava o seu primeiro bloco (as classificações), o programa2 executava o seu bloco 1, processando as suas rotinas de abertura e menu paralelamente ao programa1, ganhando dessa forma um bom tempo. O fragmento de código do programa2 a seguir mostra a transição do seu bloco 1 para o bloco 2:

```
OK=`cat pipe1`
if [ $OK = va ]
then
  ...
  Rotina de impressão
```

Tabela 1

	Programa1	Programa2
Bloco 1	Rotina de classificação de três grandes arquivos	Rotina de abertura e geração de menus
Bloco 2	Acertos finais e encerramento	Impressão dos dados classificados pelo programa 1

```
...
else
    exit 1
fi
```

Após a execução de seu primeiro bloco, o programa2 passará a "ouvir" o pipe1, ficando parado até que as classificações do Programa1 terminem, testando a seguir a mensagem passada pelo pipe1 para decidir se os arquivos estão íntegros para serem impressos ou se o programa deverá ser descontinuado. Dessa forma é possível disparar programas de forma assíncrona e sincronizá-los quando necessário, ganhando bastante tempo de processamento.

Bloqueio de arquivos

Suponha que você tenha escrito um CGI (*Common Gateway Interface*) em Shell Script para contar quantos hits uma determinada URL recebe e a rotina de contagem está da seguinte maneira:

```
Hits="$(cat page.hits 2> /dev/null)" || Hits=0
echo $((Hits=Hits++)) > page.hits
```

Dessa forma, se a página receber dois ou mais acessos simultâneos, um ou mais poderá ser perdido, bastando que o segundo acesso seja feito após a leitura do arquivo `page.hits` e antes da sua gravação, isto é, após o primeiro acesso ter executado a primeira linha do script e antes de executar a segunda.

Listagem 2: contahits.sh

```
01 #!/bin/bash
02
03 PIPE="/tmp/pipe_contador" # arquivo named pipe
04 # dir onde serao colocados os arquivos contadores de cada pagina
05 DIR="/var/www/contador"
06
07 [ -p "$PIPE" ] || mkfifo "$PIPE"
08
09 while :
10 do
11     for URL in $(cat < $PIPE)
12     do
13         FILE="$DIR/$(echo $URL | sed 's,./,,')"
14         # quando rodar como daemon comente a proxima linha
15         echo "arquivo = $FILE"
16
17         n="$(cat $FILE 2> /dev/null)" || n=0
18         echo $((n=n+1)) > "$FILE"
19     done
20 done
```

Então, o que fazer? Para resolver o problema de concorrência, vamos utilizar um named pipe. Criamos o script na **listagem 2** que será o daemon que receberá todos os pedidos para incrementar o contador. Note que ele vai ser usado por qualquer página no nosso site que precise de um contador. Como apenas este script altera os arquivos, não existe o problema de concorrência.

Este script será um *daemon*, isto é, rodará em segundo plano. Quando uma página sofrer um acesso, ela escreverá a sua URL no pipe. Para testar, execute este comando:

```
echo "teste_pagina.html" > /tmp/pipe_contador
```

Para evitar erros, em cada página a que quisermos adicionar o contador acrescentamos a seguinte linha:

```
<!--#exec cmd="echo $REQUEST_URI > /tmp/pipe_contador"-->
```

Note que a variável `$REQUEST_URI` contém o nome do arquivo que o browser requisitou. Esse exemplo é fruto de uma troca de idéias com o amigo e mestre em Shell Tobias Salazar Trevisan, que escreveu o script e colocou-o em seu excelente site (www.thobias.org). Aconselho a todos os que querem aprender Shell a dar uma passada lá.

Você pensa que o assunto named pipes está esgotado? Enganou-se. Vou mostrar um uso diferente a partir de agora.

Substituição de processos

Vou mostrar que o Shell também usa os named pipes de uma maneira bastante singular, que é a substituição de processos (*process substitution*). Uma substituição de processos ocorre quando você põe um `<` ou um `>` grudado na frente do parêntese da esquerda. Digitar o comando:

```
$ cat <(ls -l)
```

Resultará no comando `ls -l` executado em um sub-shell, como normal, porém redirecionará a saída para um named pipe temporário, que o Shell cria, nomeia e depois remove. Então o `cat` terá um nome de arquivo válido para ler (que será este named pipe e cujo dispositivo lógico associado é `/dev/fd/63`). O resultado é a mesma saída que a gerada pelo `ls -l`, porém dando um ou mais passos que o usual. Pra que simplificar?

Como poderemos nos certificar disso? Fácil... Veja o comando a seguir:

```
$ ls -l >(cat)
l-wx-- 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É... Realmente é um named pipe. Você deve estar pensando que isto é uma maluquice de nerd, né? Então suponha que você tenha dois diretórios, chamados `dir` e `dir.bkp`, e deseja saber se os dois são iguais. Basta comparar o conteúdo dos diretórios com o comando `cmp`:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) || echo backup furado
```

ou, melhor ainda:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) >/dev/null || echo backup furado
```

Este é um exemplo meramente didático, mas são tantos os comandos que produzem mais de uma linha de saída que ele serve como guia para outros. Eu quero gerar uma listagem dos meus arquivos, numerando-os, e ao final mostrar o total de arquivos no diretório corrente:

```
while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done < <(ls)
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Tá legal, eu sei que existem outras formas de executar a mesma tarefa. Usando o comando `while`, a forma mais comum de resolver esse problema seria:

```
ls | while read arq
do
    ((i++)) # assim nao eh necessario inicializar i
    echo "$i: $arq"
done
echo "No diretorio corrente (`pwd`) existem $i arquivos"
```

Ao executar o script, tudo parece estar bem, porém no comando `echo` após o `done`, você verá que o valor de `$i` foi perdido. Isso deve-se ao fato desta variável estar sendo incrementada em um sub-shell criado pelo pipe (`|`) e que terminou no comando `done`, levando com ele todas as variáveis criadas no seu interior e as alterações lá feitas por variáveis criadas externamente.

Somente para te mostrar que uma variável criada fora do sub-shell e alterada em seu interior perde as alterações feitas quando o sub-shell se encerra, execute o script a seguir:

```
#!/bin/bash
LIST="" # Criada no shell principal
ls | while read FILE # Inicio do subshell
do
    LIST="$FILE $LIST" # Alterada dentro do subshell
done # Fim do subshell
echo $LIST
```

No início deste exemplo eu disse que ele era meramente didático porque existem formas melhores de fazer a mesma tarefa. Veja só estas duas:

```
$ ls | ln
```

ou então, usando a própria substituição de processos:

```
$ cat -n <(ls)
```

Um último exemplo: você deseja comparar `arq1` e `arq2` usando o comando `comm`, mas esse comando necessita que os arquivos estejam classificados. Então a melhor forma de proceder é:

```
$ comm <(sort arq1) <(sort arq2)
```

Essa forma evita que você faça as seguintes operações:

```
$ sort arq1 > /tmp/sort1
$ sort arq2 > /tmp/sort2
$ comm /tmp/sort1 /tmp/sort2
$ rm -f /tmp/sort1 /tmp/sort2
```

Pessoal, o nosso papo de botequim chegou ao fim. Curti muito e recebi diversos elogios pelo trabalho desenvolvido ao longo de doze meses e, o melhor de tudo, fiz muitas amizades e tomei muitos chopes de graça com os leitores que encontrei pelos congressos e palestras que ando fazendo pelo nosso querido Brasil. Me despeço de todos mandando um grande abraço aos barbudos e beijos às meninas e agradecendo os mais de 100 emails que recebi, todos elogiosos e devidamente respondidos. À saúde de todos nós: Tim, Tim.

- Chico, fecha a minha conta porque vou pra casa! ■

SOBRE O AUTOR

Julio Cezar Neves é Analista de Suporte de Sistemas desde 1969 e trabalha com Unix desde 1980, quando participou do desenvolvimento do SOX, um sistema operacional similar ao Unix produzido pela Cobra Computadores. É autor do livro Programação Shell Linux, publicado pela editora Brasport. Pode ser contatado no e-mail julio.neves@gmail.com